第2章

C言語から直接生成したステート・マシン記述を用いたFFT回路の設計

Design Wave設計コンテスト2007 Professional 部門第1位

石井康雄

Design Wave 設計コンテスト2007 のProfessional部門第1位の設計を紹介する．乗算器の数を減らすために，仕様書で示されたRADIX-4方式ではなく，RADIX-4とRADIX-2を組み合わせたSplit-RADIX方式を採用した．また，加算器に対して規模の大きい乗算器の使用効率を高めるために，スケジューリングを行い，そのステート・マシンを生成するソフトウェアをC言語で開発した．（編集部）

昨年は学生として参加したこのコンテストですが，今回は社会人として参加することになりました．目標は昨年争った学生達と沖縄で再会することです注1．

さて，今回の課題はFFTです．私は組み込み技術者というよりは計算機技術者なので，組み込み機器におけるFFTの利用よりはFFTE(1)などをはじめとした科学技術計算の話の方が身近に感じられます．そこで，低コストFPGA上への実装をプロトタイプとして最終的に高性能なFPGAやASIC上で実装し，科学技術計算に利用できる高性能FFTアクセラレータを作成するといったストーリを作れないものかと考えて設計を開始しました．

科学技術計算におけるアクセラレータというアプローチが近年注目を浴びています．GRAPEプロジェクト(2)のように完全な専用計算機もあれば，PlayStation 3に採用されているCELLプロセッサ(3)のSPE（Synergistic Processor Element）のように汎用的なSIMD（single instraction stream-multiple data stream）エンジンもあります．特に，GRAPE-7はFPGAで実装されていますから，FPGAによる科学技術計算は今日では選択肢の一つとして考えられていると言ってよいと思います注2．

数値的な目標はさておいて，当然，科学技術計算をさせるのだから，64ビットの浮動小数点演算などが必要になるだろう，などと考えながら，設計の方針を決めていきました．

注1：昨年に引き続き参加している学生がおり，非常に楽しいコンテストだった．

注2：2007年3月22日にMaxwell(4)というFPGAベースのスーパコンピュータが発表された．

1．設計の方針と目標

科学技術計算に使えるアクセラレータなどと大それた目標を立てたものの，実際に作るのはプロトタイプもどきです．低コストFPGAにFFT回路を実装して具合をみることにしました．

プロトタイプからの将来的な拡張もにらんで設計する必要があります．そこで今回は，設計の目標を以下のように定めました．

- 回路の小規模化
- 高性能化（高効率化）
- 可搬性の高さ

● 乗算器を削減して回路規模を抑える

回路の小規模化は，コスト削減につながります．回路が小規模であれば，ダイが小さくなるので，コスト削減には直接的な効果があります．また，一般に半導体製造においては，回路規模が下がるほど歩留まりが上がるので，回路規模は小さいに越したことはありません．回路の小規模化

KeyWord　Split-RADIX，FFT，FPGA，バタフライ演算，複素数，三番地文，DAG，スクラッチパッド・メモリ

は必須の目標と考えました.

具体的には，乗算数の削減という最適化をしました．一般に，固定小数点や浮動小数点の演算器では，乗算器は加減算器と比較して多くの資源を必要とする傾向にあります注3．このため，アルゴリズムの変更により乗算の回数を減らすことで，必要とする乗算器を削減することは，回路規模や消費電力の削減に有効です．しかも，将来的にビット幅を増やす可能性が高い場合に，乗算器がビット幅の拡張により最も資源量が増えるものであるという事実も見逃せません.

● 低コストFPGA向けを想定して高い効率を目指す

また，最終的にアクセラレータとして動作してほしいので，ある程度の演算速度，演算効率を実現することを目標とします．専用アクセラレータは汎用計算機で出せない性能注4を出すことが最低条件なので，これも当然の目標です.

とはいえ，低コストFPGAで高性能・高効率を両方達成するのは困難ですから，高性能はひとまず据え置いて高効率を目指すことにします．加算器と乗算器のスループットが等しい汎用プロセッサの環境では，演算器の利用効率注5が62.5％を超えることがないので，63％以上の演算器の利用率を目標としました.

また，演算性能はFPGAの性能も考慮し，演算回路の内部クロック100MHzを目標としました．低コストFPGAでこの程度動くなら，高価格帯で150MHz～300MHz程度，ASIC注6にすれば300MHz～500MHz程度の動作周波数が得られるでしょう．また，演算器や内部メモリも大量に搭載できますから，専用計算機として実現する価値のあるものができそうです.

● 仕様変更を考慮して可搬性を高める

拡張性の高さも追及しなくてはなりません．今回の設計はプロトタイプという位置づけであれば，いったん実用化した後にさらなる拡張の要求が必ずあるわけです．この要求としては，例えばデータ幅の増大（固定小数点数から浮動小数点数の利用への変更），FFTに利用する点数の増大（64点から4096点など）などが挙げられます．このような要求に対して最小のコスト（VHDLファイルの一部の定数値の変更など）で対処できるようにデータ・パスを設計したいと考えました.

● コンテスト向けにおもしろく

これらの目標を達成するために，アルゴリズムとデータ・パスの実装方法を再検討して設計を行うことにしました．また，コンテストという特性もあるので，トレードオフで迷うことがあった場合には，おもしろい方を選ぶということも設計方針に付け加えています.

2．アルゴリズムの検討

ここでは，コンテストで採用したアルゴリズムと，その背景に関して説明します.

まず，最適化は大きく二つに分かれています．一方が複素数乗算のアルゴリズムの最適化で，もう一方がFFTのバタフライ演算のアルゴリズムの最適化です．それぞれに対して注目し，可能な限り乗算回数を減らすように最適化を行いました.

● 複素数計算のアルゴリズム

当たり前の話なのですが，FFT中に発生する乗算は，基本的にはすべて回転因子との複素数乗算です注7．これを踏まえて最適化を行っていきましょう.

一般に複素数（x_r, x_i）と回転因子（w_r, w_i）の乗算結果（y_r, y_i）は，以下のように計算されます.

$$y_r = x_r \cdot w_r - x_i \cdot w_i$$
$$y_i = x_r \cdot w_i + x_i \cdot w_r$$

これは皆さんご存知の数学の教科書に載っている方法です．この計算では，一つの複素数乗算に対して4回の実数乗算が必要です．しかし，この計算の形式を変更すると，乗算の回数を3回に減らすことができます注8.

$$tmp = (x_r + x_i) \cdot w_r$$
$$y_r = tmp - x_i \cdot (w_r + w_i)$$
$$y_i = tmp - x_r \cdot (w_r - w_i)$$

注3：$N \times N$乗算器の乗算器は$O(N^2)$以上の資源を必要とする．しかし，$N+N$の加算器は$O(N)$～$O(N \log N)$程度で実現されることが多い.
注4：ここでの性能とは消費電力あたりの演算性能などであってもよい.
注5：このレポート中では演算全体にかかった時間（サイクル数）で演算器が実際に演算した時間を除したもので定義する.
注6：ほんとうに作るとなれば，米国Altera社のHardCopyのようにFPGAからASICへの設計パスが存在するので，それを利用するのがよいだろう.
注7：RADIX-8の場合にはスカラ値との積も含まれる.
注8：正確には2回の加算と4回の乗算を3回の加算と3回の乗算に置き換えることになる.

この計算中に出てくる値のうち$w_r + w_i$と$w_r - w_i$は回転因子の和と差なので，事前に計算可能です．この値をあらかじめ計算し，ROM領域に保持することで3回の加減算と3回の乗算で実現可能です．

しかし，この方法を知らなかった方も多いと思います．なぜでしょうか．これには，いくつかの理由があります．

- FFTではもともと実数乗算の回数が実数加算の回数と比較して少ない[注9]
- 多くの汎用のプロセッサ（x86など）では乗算と加算のスループットが等しい[注10]．
- 加減算はけた落ちの危険が高いので多用したくない（浮動小数点演算の場合）．

もともと演算回数が少なく，遊んでいる乗算器の利用効率を上げることにも意味がないため，この最適化を行う必要はなかったわけです．しかも，複素数の加減算の回数と乗算の回数が等しい場合には，4回の加減算と4回の乗算となり，汎用プロセッサとしては具合が良いアルゴリズムになっているという背景もあります．

しかし，今回のように自由に演算リソースが割り当てられる場合には，加算器を増やすことで乗算数削減のメリットを享受することができます．

けた落ちの危険については，ほんとうは考慮する必要があるのですが，固定小数点には桁落ちの心配がないことと，コンテスト用のカスタマイズということで目をつぶることにしました．実際には適用するアプリケーションと必要な精度などを考慮して，この最適化を適用するかどうかを考える必要があります．

以上の考察から，今回の設計ではこの乗算回数を間引いた複素数乗算方式を採用することにしました．

注9：一般的なFFT演算のアルゴリズムで，実数加算と実数乗算の比率は，おおよそ4：1（表1参照）

注10：ここでの加減乗除は浮動小数点数を仮定している．

注11：例えば，HPC分野で利用されるFFTEでは基底2，3，4，5，8のFFTを混合して計算をする．

● FFTバタフライ演算

繰り返しになりますが，FFT演算の中で発生する乗算は，基本的にはすべて回転因子との複素数乗算です．回転因子は，どこで，どの回転因子を適用するかがFFTの問題サイズで事前に決まっているため，これを利用して一部の回転因子との複素数乗算を取り除いたり，スカラ乗算へ変更することが可能です．これはバタフライ演算のバリエーションを増やすことで可能になります．

また，FFTの基数の増加は，同様に乗算の回数を削減する効果があります．課題で示されたアルゴリズムはRADIX-4方式で，最も利用されている基数のひとつです．もちろん，これを単純に拡張したRADIX-8方式なども乗算回数を減らす手段として知られているのですが[注11]，今回はさらなる最適化を目指し，Split-RADIXという手法を用います．

このSplit-RADIX方式では，RADIX-4演算とRADIX-2演算を混ぜることにより，RADIX-8方式以上に乗算回数を削減することが可能となるアルゴリズムです．一般のバタフライ演算と異なり，バタフライの形状をL字型にとることによって，無駄な複素数乗算を省き，加算も可能な限り削減できます．

しかし，Split-RADIX方式のFFT演算にはひとつ非常に重大な問題点があります．それは，ループ構造が非常に複雑になるということです．一般に複雑なループ構造は，プロセッサの実行効率を著しく下げてしまいます．このことが，Split-RADIX方式がメジャーになれない要因となっています．

今回はこの問題を完全なループ展開を併用することで解決しました．これは汎用のFFTプログラムでなく，N点限定のFFTアクセラレータだからできることだと思います．

具体的には，64点のFFTはバタフライの数やFFTの基数を決定すると，演算の順序などは一意に定まるという性質を利用します．一意に定まっているわけですから，そのループ構造に沿ってすべてのループを展開すれば，問題だった複雑なループ構造は消えてしまいます．もしも，

表1 実数演算回数の比較

方式	複素数乗算最適化あり 5-バタフライ		複素数乗算最適化なし 5-バタフライ		複素数乗算最適化なし 2-バタフライ		仕様書の方式	
	乗算	加算	乗算	加算	乗算	加算	乗算	加算
RADIX-2	264	1032	332	964	516	1026	640	1088
RADIX-4	208	976	264	920	324	930	**384**	960
RADIX-8	204	972	248	928	260	930	288	960
Split-RADIX	**196**	964	248	912	—	—	—	—

4096点のFFTを実装したい場合には，もう一度ループを展開しなおすようにします．

これで，FPGAの柔軟性とハードウェア実装ならではの最適化が完成しました．

● FFTアルゴリズムと固定小数点数乗算の回数

実際に上述のアルゴリズム変更でどれだけ乗算回数が削減できるかを**表1**に示します．

64点FFTにおいては，本設計で採用したアルゴリズムでは乗算回数が196回でした．課題で示されたアルゴリズムの乗算回数（384回）から49.0%削減できていることが確認できます．その分，加算回数が増えていますが，加算器は乗算器と比較すると「安い」資源なので目標の達成はできているのではないか，と考えています．

3．アルゴリズムの実装

ここでは，アルゴリズムをどのように実装するかを検討します．以下ではデータ・パスは汎用のものを利用し，ステート・マシンはC言語から直接生成という手法をとることにより，高性能・高効率を実現していくことにします．

● 実装方針

ここで実装方針を決めて行きます．

入出力の仕様は，仕様書で示されていたものに合わせることにしました．従って，データの入出力にそれぞれ64サイクルずつ必要となります．

ここで外部クロックを4てい倍して内部で使用し，256サイクル以内にFFT自体の計算が終わるように最適化を行うことにします．乗算回数は196回なので，計算順序を工夫すればひとつの乗算器を使いまわすFFT回路が期待できます．

● 入出力関連データ・パス

外部クロックを4てい倍した内部クロックで動作するため，クロックの差を吸収するために入力バッファと出力バッファを採用しました（**図1**）．

各バッファへの読み書きは，外部からの信号送出の開始に同期して動作させることにします．

● FFT演算メイン部のデータ・パス

Split-RADIX方式のFFT演算では，すべての乗算が加算の後に行われ，かつ，実数乗算と実数加算の比が196：964　1：5になります．このため，1個の乗算器と5個の加減算器を実装し，かつ，データ・パスは**図2**のように設計しました．

この回路のデータ・パス上には，1個の乗算器と5個の加算器を配置しています．また記憶素子は，10個のRAMブロック（BANK0～BANK9）と回転因子のROMからなります．そして，それらをクロスバー・スイッチで結合しています．

回路構成を単純化するため，各RAMブロックからは決まった演算器の決まったポートにしかデータが出せなくなっています．また，単純化のため，すべてのRAMは一つの読み出しポートと一つの書き込みポートのみとしました．RAMのポート数を削減することにより，スケジューリングの難しさは増加しますが，回路は単純化します．これにより，低消費電力で高速なデータ・パスが実現可能になります．

なお，このデータ・パス中には和積演算を行うという，変な回路があります．将来的に高性能FPGA上に実装することを踏まえるなら，積和演算回路か独立にRAMを持つ構成にしたほうがよかった[注12]と考えています．

図1　データ・パスの概要
外部クロックを4てい倍した内部クロックで動作するため，クロックの差を吸収するために入力バッファと出力バッファを採用した．
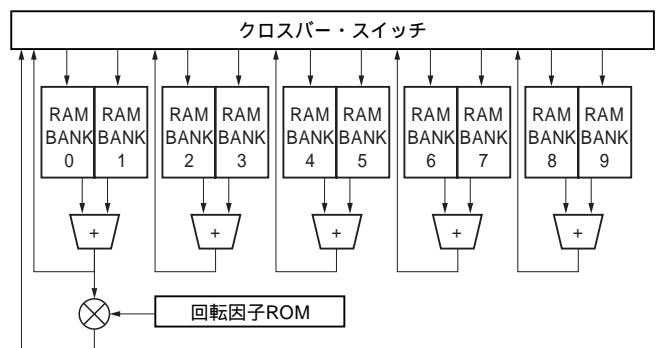
図2　FFT演算のデータ・パス
1個の乗算器と5個の加算器を配置している．記憶素子は10個のRAMブロック（BANK0～BANK9）と回転因子のROMである．全体をクロスバー・スイッチで結合している．

● 疑似命令を使って制御信号を記述

Split-RADIX方式では従来のRADIX-4方式と比較して，ループ構造が複雑になる傾向にあります．複雑なループ構造で，しかも，全部で1160回の演算が必要です．これらの演算をすべて手作業でスケジューリングするのは少々酷なため，ソフトウェアで制御信号を作成することにします．

このソフトウェア制御では，必要な信号をすべてパックした信号列を格納したROM記述を作成します．結果として，プロセッサが利用するアセンブリ・コードみたいなものができ上がってしまったので，以降はこの信号のセットを擬似命令と呼ぶことにします．この擬似命令は以下の制御用のデータを保持します．

- 各RAMブロックの読み出しアドレス
- 各RAMブロックの書き込みアドレス
- 各RAMブロックのライト・イネーブル信号
- 回転因子のROMデータ
- クロスバー・スイッチの制御信号

この信号幅はあわせて169ビットあります．

以下では，この擬似命令と前章で示したデータ・パスを利用して，どうやってFFT演算を実現していくかについて説明します．

● ステート・マシンをソフトウェアで生成

今回は，FFT向けのステート・マシンをC言語[注13]のプログラムからほぼ自動で導くことにしました．これにより，点数の増加や演算器のレイテンシ変更などがあった場合にも容易に対応できるようになるでしょう．

入力は，**リスト1**に示すような普通のC言語プログラムです．出力は，**リスト2**に示すようなVHDLによるROMの記述で，ソース・コードにそのまま貼り付けて使います．

ステート・マシンの具体的な生成手順は以下の通りです．

1) C言語で記述した実数演算をすべて単純な三番地文に置き換える
2) 64点のSplit-RADIX FFTのアルゴリズムのすべての加減算と乗算の実行履歴を抽出
3) 抽出された加減算から，変数の使い回し（副作用）を除去
4) 各変数をどのリソース（RAMのBANK）に割り当てるか決定
5) 演算の順序を保ったまま，効率的に演算が行えるようにスケジューリング
6) スケジューリング結果を出力

この手順の中で1）は完全に手作業で行い，残りを自動化しました．

自動化する個所では，前半で簡単なプロファイル生成を行いながら，擬似命令セットを作成しやすいようにC言語のコードを変換していきます．後半はハードウェアに合わせたスケジューリングです．これはコンパイラの基礎です．

● プロファイル作成とコード変換

簡単なプログラムで3）までの実行を行う場合の例を**図3**に示します．

ステップ1）すべての演算を三番地文に変更

ステップ1）では，後続の処理を簡単化するためすべての演算を三番地文（1演算の式）に変更します．

ステップ2）すべての加減算と乗算の実行履歴を抽出

ステップ2）では，ループ展開を行うために，実際にどの経路をプログラムが通るかのプロファイルを抽出します．

ステップ3）変数の使い回しを除去

ステップ3）では，変数への代入が2度発生しないように書き換えます．

注12：最近の高性能なFPGAでは積和演算器が搭載されているため．
注13：正確にはC＋＋なのだが，FFTのアルゴリズムの記述分ではC＋＋の機能をほとんど何も使っていないのでC言語と書いている．

リスト1　入力するCソース・コード

```
void fft(double ar[64], double ai[64]){
for(int i, j...) {
// ar, aiにFFT操作をする処理
// 今回はSplit-RADIX FFTを採用
// 詳細は省略
}
}
```

リスト2　出力されるVHDL記述

```
constant state_rom : state_array := (
 -- ここから一つ目の169bitのVLIW命令
( state => (
 "0000000000000000", -- RAM0の制御信号
 "0000011010000000", -- RAM1の制御信号
 "0000011010000000", -- RAM2の制御信号
 "0000011010000000", -- RAM3の制御信号
 "0000011010000000", -- RAM4の制御信号
 "0000011010000000", -- RAM5の制御信号
 "0000011010000000", -- RAM6の制御信号
 "0000011010000000", -- RAM7の制御信号
 "0000011010000000", -- RAM8の制御信号
 "0000011010000000" -- RAM9の制御信号
 ),
 wdata => "000000000"), -- 利用する回転因子データ
-- ここまで一つ目の169bitのＶＬＩＷ命令
... (以下、同様の形式で255命令並ぶ) ...
);
```

さて，ステップ3)を終えたコードでひとつ気付くことがあります．それは，このプログラムが容易に回路に変換できるということです．ループが完全に展開された状態で，変数への副作用がないのですから，Verilog HDLなら`wire`宣言と`assign`文だけを利用してこのプログラムの挙動を書くことができるのです(**リスト3**)．

● ハードウェアに合わせたスケジューリング

リスト3の回路でFFT専用回路とすると，196個の乗算器が必要になる超巨大回路ができ上がってしまいます．動作も遅いので使い物になりません．そこで，さらにこれを高速化しつつ，ひとつの乗算器で計算するためにスケジューリングを行う必要があります．

ステップ4）変数の割り当て

ステップ4)以降では，データ・パスに合わせたスケジューリングを行います．まずは変数のバンク(BANK0～BANK9)の割り付けからはじめました．なぜならすべてのデータはRAMのいずれかのバンクに存在する必要があり，かつ，各演算器はそのソースを二つの決まったバンクから読まなくてはいけないという制約があるためです．しかも，加算結果を乗算に利用する場合には，その加算の入力二つは必ずBANK0とBANK1に置く必要があるといった厳しい制約があることもあり，手ですべての制約を満たしつつスケジューリングを行うことが困難だったため，以降の処

サンプル・プログラム(簡単な積和演算の例)

```
for(i=0; i<10; i++) {
sum += a[i] * b[i];
}
```

ステップ1．すべての文を1演算(三番地文)で書くようにする．

```
for(i=0; i<10; i++) {
tmp = a[i] * b[i];
sum = sum + tmp;
}
```

手で展開

ステップ3．副作用の除去(変数の書き換えをしないように変形)

```
tmp_0 = a[0] * b[0];
sum_1 = sum_0 + tmp_0;
tmp_1 = a[1] * b[1];
sum_2 = sum_1 + tmp_1;
... (中略) ...
tmp_9 = a[9] * b[9];
sum_10 = sum_9 + tmp_9;
```

プログラムで変換

図3　前半部で行うプロファイル作成とコード変換処理の例

ステップ1)では，後続の処理を簡単化するためすべての演算を三番地文(1演算の式)に変更する．ステップ2)では，ループ展開を行うために，実際にどの経路をプログラムが通るかのプロファイルを抽出する．ステップ3)では，変数への代入が2度発生しないように書き換える．

リスト3　途中経過の出力とHDL記述の例

```
void fft(double ar[64], double ai[64]) {
 double r_0 = ar[0] . ar[32];
 double r_1 = ai[0] . ai[32];
 double r_2 = ar[0] + ar[32];
 double r_3 = ai[0] + ai[32];
 double r_4 = ar[48] . ar[16];
 double r_5 = ai[48] . ai[16];
 double r_6 = ar[16] + ar[48];
 double r_7 = ai[16] + ai[48];
 double r_8 = r_0 . r_5;

 (...中略...)
 double r_1156 = r_842 . r_942;
 double r_1157 = r_843 . r_943;
 double r_1158 = r_842 + r_942;
 double r_1159 = r_843 + r_943;
 ar[0] = r_1078;
 ai[0] = r_1079;
 ar[1] = r_1082;
 (...中略...)

 ar[62] = r_1156;
 ai[62] = r_1157;
 ar[63] = r_1074;
 ai[63] = r_1075;
}
```

Split-RADIX FFTでの1160演算を行っている個所

出力を配列に書き出している

(**a**) ステップ3 の終了後のFFT

```
module FFT (
 ar_0,...,ar_63, ai_0,...,ai_63,
 dr_0,...,dr_63,di_0,...,di_63);
 input [7:0] ar_0, ar_0,...;
                        // 64点同時入力用のポート定義
 output [13:0] dr_0, di_0, ...;
                        // 64点同時出力用のポート定義
 wire [13:0] r_0, r_1, r_2, ..., r_1159;

 assign r_0 = ar_0 . ar_32;
 assign r_1 = ai_0 . ai_32;
 ... (中略) ...
 assign r_1159 = r_843 + r_943;
 assign dr_0 = r_1078;
 ... (中略) ...
 assign dr_63 = r_1074;
 assign di_63 = r_1075;

endmodule
);
```

副作用がないため，特に何もせずHDL化することが出来る．回転因子ROMは定数で展開する．乗算のビット幅にも注意する必要がある．

(**b**) Verilog化した場合

理はソフトウェアで自動化しています．

このスケジューリングは，計算の木構造（DAG: directed acyclic graph）へのパターン・マッチングで実現しました．これもコンパイラの最適化手法としては一般的なものですから，詳しい説明は省きます．なお，このステップ4）の段階では各バンクには無限にエントリがあるとしてマッピングを行います．また，この時点では時間的にいつ計算するかも決めていません．ただ，利用するバンクを決めているだけです．

実際の処理内容の例を**図4**に示します．この例では③の計算でr_530を乗算に利用しているので，①の演算のr_521とr_524はBANK0とBANK1に配置する必要があります．また，②はr_530を利用していますが，乗算側とは別のレジスタ上にマップされるので，それを踏まえてスケジュールする必要があります．そして，④に含まれるr_529とr_531は計算のペアなので，必ずBANK2とBANK3などのようにペアになるバンクを割り当てる必要があるということになります．

ステップ5）演算のスケジューリング

どのバンクのペアで計算するかは決まったので，ステップ5）ではいつ計算するかを決定します．具体的にはある演算の入力側のオペランドが揃った命令を加算器に対して発行し，利用し終わった値を格納しているエントリがあれば，そのエントリを解放するということを行います．変数の生存グラフを解きながら，計算順序を決めるわけです．例えば，r_530は①の命令発行後に生成され，②と③の命令が発行された後には，不要となるためメモリ資源を解放します．また，②は①の3クロック後に発行可能になります．また，もしも命令を発行する場合には，そのときに必要なRAMのリード・ポートやライト・ポートが空いている必要があるという制約があります．

ステップ4）とステップ5）の処理は基本は乗算優先のマッピングです．乗算器とその直前の加算器はどうしても込みやすくなるので最初に優先的に計算するようになっています．こうして，いったん無限のエントリがある仮想的なRAMに対して変数をマッピングし，その後，変数の生存グラフを解きながら，実メモリへのマッピングを行うという操作は，一般的なコンパイラの最適化技法のひとつで，GCCなどでも利用されています．

ステップ6）VHDLソース・コードの出力

ステップ6）ではVHDLの`constant`文の形式でこのステート・マシンの内容を出力するように設定しました．こ

（**a**）処理手順

（**b**）ステップ4，5での操作を可視化（DAG構造化）したもの
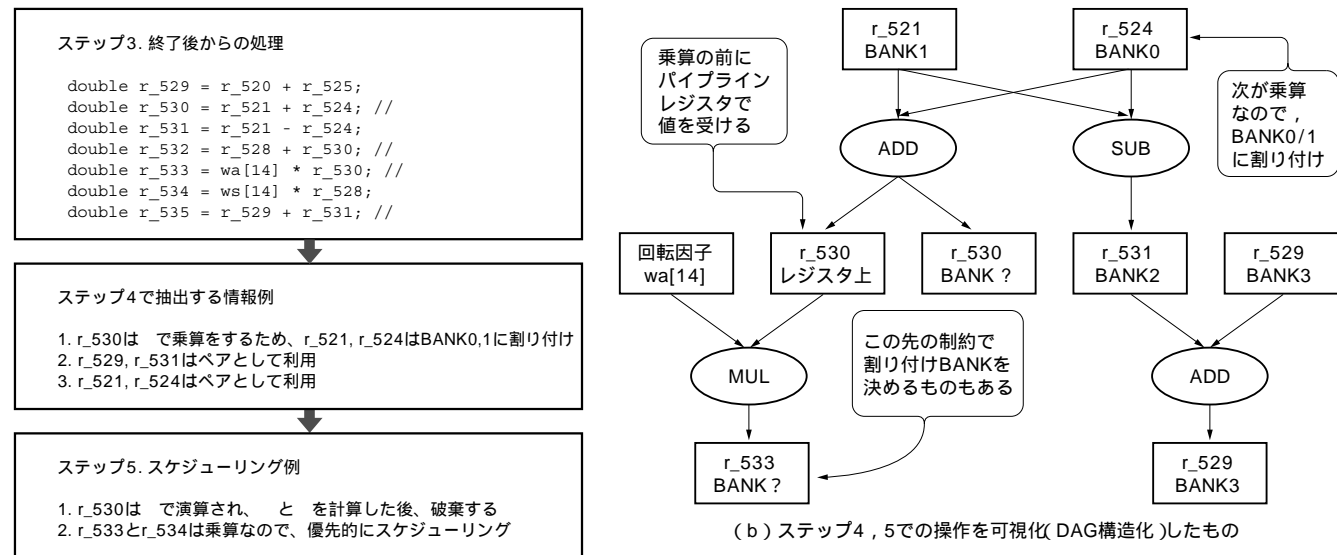
図4　後半部で行うハードウェアに合わせたスケジューリングの例

ステップ4）では，例えば③の計算でr_530を乗算に利用しているので，①の演算のr_521とr_524はBANK0とBANK1に配置することを決める．ステップ5）ではいつ計算するかを決定する．

の出力は直接VHDLファイルに貼り付けることができます(今回の場合には出力が4000行くらいある).

このプログラム中では演算器のレイテンシなどをパラメータ化しているので,例えば演算器を深いパイプラインで切るなどといった処理は非常に高効率に行うことができます.

ここまでの議論から,この半自動でのステート・マシン設計は**図2**のデータ・パスを持つVLIWプロセッサに対するマシン・コード生成といってもよいだろうと私は考えています注14.

結果としては,無事このコンパイラで256サイクル以内の演算を定義することができました.おそらく,乗算優先のポリシを貫いたことがよかったのだと考えています.

● 演算器の利用効率

256サイクルの中で,すべての演算器の利用率注15の平均値は75.5%となり,75%を超えました.これは一般的なCPUでは絶対に実現することができない効率です注16.なぜならば,加算器と乗算器の数が等しいCPUでは乗算器の利用効率が25%程度になってしまうため,演算器全体の利用効率が63%を超えることはないためです.以上から,少なくとも汎用のプロセッサを超越する高効率のアクセラレータの設計が実現できているといえるでしょう.

表2 評価環境

ターゲットFPGA	XC3S200-5PQ208
合成環境	Xilinx ISE WebPACK 8.2.03i
シミュレーション環境	ModelSim Xilinx Edition III 6.0a
制約	なし
コンパイル・オプション	すべてDefault値を利用

4.評価

ここでは作成したVHDLファイルを合成し,その評価を行います.なお,手持ちのFPGAボード(米国Xilinx社のXC3S50)に実装できなかったため,評価は論理合成とシミュレーションで行いました注17.

評価環境は**表2**に示した通りです.また,合成結果は**表3**に示したようになりました.目標とする1乗算器,内部動作100MHzを実現することができていることが分かります.なお,内部は4てい倍のクロックで動作しているため,外部クロックは25MHzと比較的低速になっています注18.また,5個のメモリ・ブロック(Block RAM)を利用していますが,各メモリ・ブロックは半分の容量しか利用していません.これはブロックRAMの最小エントリ数が512であるためで,今回の設計では256エントリしか必要ないため,このような設計となっています.ここも最適化の可能な個所ではあるのですが,時間の都合でこのままにしています.

動作確認はシミュレーションで行いました(**図5**).内部クロックを4てい倍しているため,クロックごとにデータを与えられるスループットを実現していることが分かりま

注14:ハードウェアとソフトウェアとの強調は,今後ますます重要になるテーマであり,64点FFTという問題の落としどころとして,このような切り口で設計した点を評価していただけると幸いである.
注15:演算器を利用したサイクル数を全サイクル数で割った値を想定している.今回は6個の演算器のそれぞれに対してこの値を求めた上で,その平均を評価対象とした.
注16:もちろん,専用にカスタマイズしたプロセッサではこの限りではない.TeraRecon社のグラフィックス・アクセラレータなどはその一例である(2).ちなみに,このプロセッサもVLIWの命令セットだったようで,思想は似ている気がする.
注17:こんなことなら豪華な基板を借りておけばよかった,と後悔している.
注18:近年のノイズなどを考慮した高密度基板設計の大変さを考えると外部信号を低速にするのは悪いアイデアではないと考えている.わたしの技術で100MHz超のはんだ付けなどができる気がしなかったというのもある.

表3 合成結果

	64点FFT演算回路		50入力XOR	
	使用量	使用率	使用量	使用率
LUT数	2,096	54%	17	1%
フリップフロップ数	402	10%	0	0%
スライス数	1,170	60%	13	1%
乗算器数	1	8%	0	0%
ブロックRAM	5	41%	0	0%
DCM	1	25%	0	0%
ゲート数換算値	411,965	8,246UNITAREA	2,448	49UNITAREA
クリティカル・パス	9,605ns	4,756UNITDELAY	12,117ns	6UNITDELAY
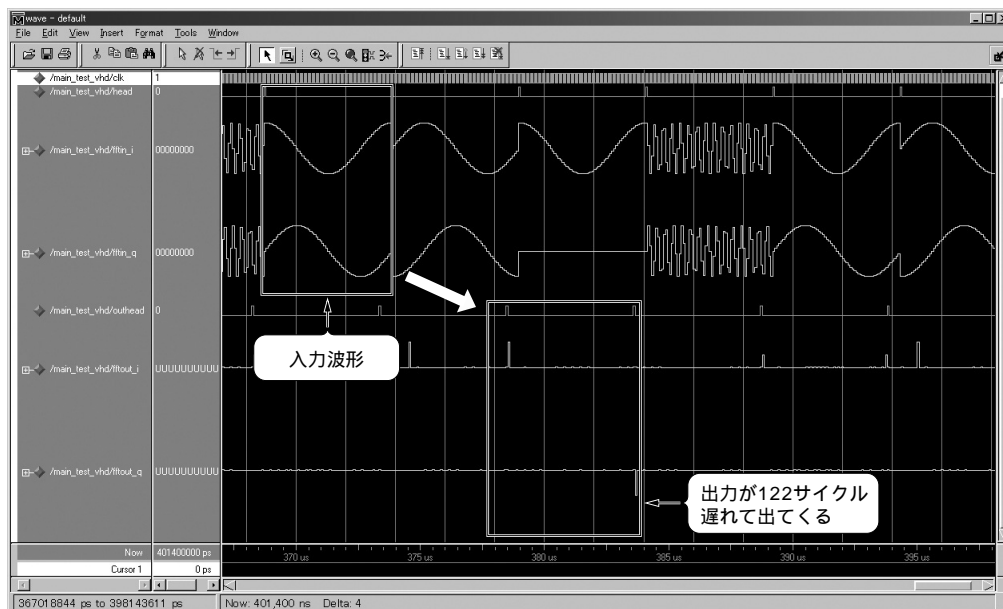
図5
シミュレーション結果
内部クロックを4てい倍しているため，クロックごとにデータを与えられるスループットを実現していることが分かる．

す．図はサンプルの入力に対する出力ですが，そのほかのサンプルに対しても期待通りの出力が得られることを確認しています．

5．さらなる最適化を目指して

これまで説明してきた回路は実際にコンテストに提出したものです．しかし，最適化のアイデア自体はまだたくさんあります．

● パイプライン処理

一つ目はパイプライン処理です．実は，私の設計能力の関係でコンパイラを書くことに手間取ってしまい，データ・パスの最適化が間に合っていませんでした．これを改善したものを**表4**に示します．低コストFPGAで150MHzを超えていますから，まあまあよいところまで攻めたのではないだろうかと考えています．

このときにコンパイラに加えた主な修正は，ADD_LATENCYという加算のレイテンシを定義する変数を1から3に変更させただけです．狙い通り，汎用的に動作するものができていたのだな，と感動してしまいました．

表4　パイプライン処理を行った場合の合成結果

	利用数	利用率
LUT数	2,090	54%
フリップフロップ数	878	22%
スライス数	1,368	71%
乗算器数	1	8%
ブロックRAM	5	41%
DCM	1	25%
ゲート数換算値	415,362	8,251 UNITAREA
クリティカル・パス	5,904 ns	2,923 UNITDELAY

● RAM制御の最適化

二つ目はRAMの制御の最適化です．そもそもバタフライ演算における加減算は，必ず加算と減算をセットにして行います．そこで，これを同時に行いやすいようにデータ・パスを設計し直してみました．

最適化したデータ・パス自体はすぐに考えることができて，**図6**のようになります．ここを最適化することができればRAMのバンク数を減らすことができます．実は，このアイデア自体は初期の頃からあったのですが，コンパイラ側の設計で時間を取られてしまい，結局，実装まで到達できませんでした．

● RAM容量の最適化

三つ目はRAMの容量の最適化です．今回の設計でRAMは10バンクで合計320エントリ分確保していたのですが，実はそのうちの6割程度しか使っていません．これは各バンクに必要なRAMのエントリ数が20エントリ程度だったから，ということが背景にあります．この100％使わないRAMのエントリために資源を割くのもばかばかしいので
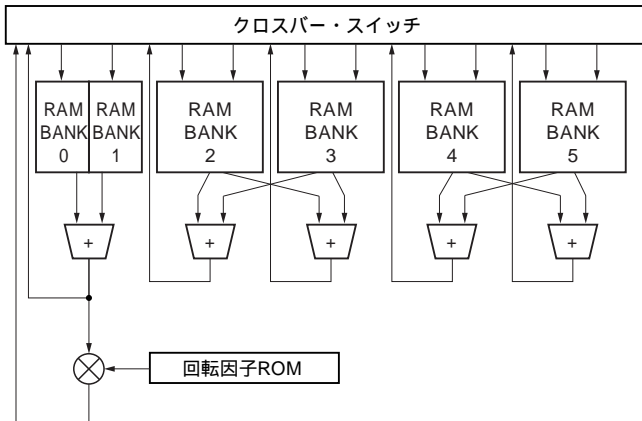
図6　RAM制御の最適化のアイデア
バタフライ演算における加減算は，加算と減算をセットにして行うので，これを同時に行いやすいようにデータ・パスを設計し直す．
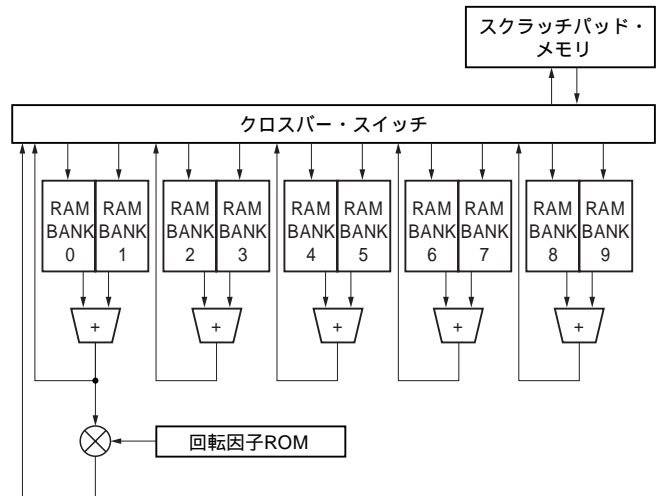
図7　RAM容量の最適化のアイデア
各RAMのエントリを16程度で固定してしまい，あふれたデータをスクラッチ・パッドに保存する．

何とか改善したいものです．

アイデアとしてはCELLプロセッサのLocal Storeのようなスクラッチパッドを搭載することが挙げられます．**図7**のように，各RAMのエントリを16程度で固定してしまい，あふれたデータをスクラッチパッドに保存しておくということになります．これもスピル処理としてよく知られたものです．これを実現できればメモリの利用効率は向上することが期待できます．

● 多点FFTのサポート

四つ目は多点FFTのサポートです．現在の実装をそのまま，多点のFFTに適用した場合には，ステート・マシンが肥大化してしまいます．これは状態の再利用で解決できると考えています．

例えば，FFTは**リスト4**のような再帰的なアルゴリズムでも記述できます．この再帰で呼ぶ先のアルゴリズムを一定点数以下になったら変更するという手法をとればよいのではないかと考えています．

● 改善点はまだまだ多い

ほかにも，沖縄で開催されたコンテスト発表会の参加者の方からクロスバーをやめてリング・バスにしたらどうか，といった指摘を受けました．「やれること，やるべきこと」を整理して列挙すると，やることだらけだな，と改めて感じます．もっと勉強をして，こうしたことを実現していけるエンジニアになっていきたいものです[注19]．

リスト4　再帰処理を利用したFFTプログラム

```
void fft(int n, double theta, complex a[], complex
tmp[])

{
 int nh, j;

 if (n <= 1) return;
 nh = n / 2;
 for (j = 0; j < nh; j++) {
  complex x = a[j] . a[nh + j];
  complex w = (cos(theta * j), sin(theta * j));
  tmp[j] = a[j] + a[nh + j];
  tmp[nh + j] = x * w;
 }

 fft(nh, 2 * theta, tmp, a); // 再起呼び出し箇所
 fft(nh, 2 * theta, &tmp[nh], a); // 再起呼び出し箇所

 for (j = 0; j < nh; j++) {
  a[2 * j] = tmp[j];
  a[2 * j + 1] = tmp[nh + j];
 }
}
```

6．コンテスト発表会に参加して

沖縄で開催されたコンテスト発表会において，学生の発表を聞いて感じたことを紹介します．

まず，レベルが非常に高いと感じました．京都大学チームのマルチコアFFTももちろんなのですが，ほかの参加チームでも200MHz～400MHzで回路を動作させたチームがありました．つい先日まで233MHz動作のPentium IIを搭載した パソコンを利用[注20]していた私は，大いに驚かさ

注19：これが一番のFuture Workだろう．
注20：サーバとしての利用だが．

れました．また，20歳前後の非常に若い学生が果敢にチャレンジしているさまも印象に残っています注21．こうした後輩達に負けないためにももっと精進しなくては，と良い刺激を受けたコンテストでした．

7．まとめ

今回のコンテストでは限られたリソースを有効活用するための最適化アルゴリズムを採用し，それに基づくFFT演算回路の設計を行いました．この提案アルゴリズムでは乗算の回数を極限まで削減することによる演算回路のリソース量を削減することを目指しました．

また，データ・パス構成を汎用的な構成にして，ステート・マシンの記述をソフトウェアによる自動生成で作成し，急な仕様変更などにも柔軟に対応できることを目標として設計を行いました．拡張後に最良の設計となるように設計したため，現在の設計は現在の仕様の下では最良ではないものになっています注22．

その結果として，100MHzでメイン部分が動作をし，演算器の利用効率75％超を実現するFFT演算回路を設計することができました．このことは，ソフトウェア・ハードウェアの協調設計の強みをアピールすることができたものと考えています．

また，隠れたチャレンジとして，今回の設計はすべて無償で利用できるソフトウェアを利用して作成してみました．一昔前は，「EDAツールのライセンスがないからHDLで設計することができない」などといって嘆いた時代だったはずですが，今では，誰でも簡単にプログラム感覚で回路が書ける時代になっているということを改めて感じることになりました．

入社以来，アーキテクチャから実装まですべてを自由に作る機会というものが存在しなかったので，久しぶりに自分の裁量で上から下まですべてを設計し，非常に楽しく，かつ，貴重な経験をすることができました．

注21：私が20歳の頃は，まだHDLを書いたことがなかった．
注22：今回の構成よりも，より小さく，より速い単純ですなおな64点FFT回路を書くのは簡単．しかし，私がひねくれ者なので，こういった実装になった．

参考・引用*文献

(1) FFTE；A Fast Fourier Transform Packages，http://www.ffte.jp/
(2) the Grape project，http://grape.astron.s.u-tokyo.ac.jp/grape/
(3) Cell Broadband Engine，http://cell.scei.co.jp/
(4) FPGA High Performance Computing Alliance，http://www.fhpca.org/press_maxwell.html
(4) 小林芳直；ディジタル・ハードウェア設計の基礎と実践--高性能高信頼システムを開発するための定石，CQ出版社．
(5) 48個のALUが並列動作する実行性能10.2GFLOPSの画像処理プロセッサを開発，Design Wave Magazine，2006年5月号．

いしい・やすお
NEC　コンピュータ事業部
y-ishii@bc.jp.nec.com

<筆者プロフィール>
石井 康雄．スーパコンピュータの開発に興味を持っている入社1年目．専門はプロセッサ・アーキテクチャ（特に分岐予測方式）．最近は，学位を持っておいた方が良いかなと考えて大学に戻ることも検討しているが，やっぱり生活できなくなりそうで悩んでいる．「業務」と「研究」と「あそび」と「自己啓発」の中心で「おもしろい仕事がしたい！」と叫ぶ25歳．